**北京科技大学实验报告**

学院： 计通学院 专业： 计算机科学与技术 班级： 计172

姓名： 金玉卿 学号： 41724051 实验日期： 2020年 6月 27 日

**实验名称：**

PL/0编译程序的实现

**实验目的：**

通过实现对PL/0源代码的词法分析，语法分析，语义分析，生成Pcode代码并解释运行，了解编译程序的编译过程。

**实验仪器：**

* 操作系统：win10
* 开发工具：eclipse JRE: 1.8
* 编程语言：java

**实验原理：**

PL/0语言是PASCAL语言的子集，它的编译程序是一个编译解释执行系统。PL/0的目标程序为假想栈式计算机的汇编语言，与具体计算机无关。其编译过程采用一趟扫描方式，以语法分析程序为核心，词法分析和代码生成程序都作为一个独立的过程，当语法分析需要读单词时就调用词法分析程序，而当语法分析正确需要生成相应的目标代码时，则调用代码生成程序。此外，用表格管理程序建立变量、常量和过程标示符的说明与引用之间的信息联系。用出错处理程序对词法和语法分析遇到的错误给出在源程序中出错的位置和错误性质。

PL/0文法的EBNF描述：

<程序> ::= <分程序>.

<分程序> ::= [<常量说明部分>][变量说明部分>][<过程说明部分>]<语句>

<常量说明部分> ::= const<常量定义>{,<常量定义>};

<常量定义> ::= <标识符>=<无符号整数>

<无符号整数> ::= <数字>{<数字>}

<标识符> ::= <字母>{<字母>|<数字>}

<变量说明部分>::= var<标识符>{,<标识符>};

<过程说明部分> ::= <过程首部><分程序>;{<过程说明部分>}

<过程首部> ::= procedure<标识符>;

<语句> ::= <赋值语句>|<条件语句>|<当型循环语句>|<过程调用语句>|<读语句>|<写语句>|<复合语句>|<重复语句>|<空>

<赋值语句> ::= <标识符>:=<表达式>

<表达式> ::= [+|-]<项>{<加法运算符><项>}

<项> ::= <因子>{<乘法运算符><因子>}

<因子> ::= <标识符>|<无符号整数>|'('<表达式>')‘

<加法运算符> ::= +|-

<乘法运算符> ::= \*|/

<条件> ::= <表达式><关系运算符><表达式>|odd<表达式>

<关系运算符> ::= =|<>|<|<=|>|>=

<条件语句> ::= if<条件>then<语句>[else<语句>]

<当型循环语句> ::= while<条件>do<语句>

<过程调用语句> ::= call<标识符>

<复合语句> ::= begin<语句>{;<语句>}end

<重复语句> ::= repeat<语句>{;<语句>}until<条件>

<读语句> ::= read'('<标识符>{,<标识符>}')‘

<写语句> ::= write'('<标识符>{,<标识符>}')‘

<字母> ::= a|b|...|X|Y|Z

<数字> ::= 0|1|2|...|8|9

一个经典的编译程序一般包括7个部分：词法分析，语法分析，语义分析及代码生成，代码优化（可省略），代码执行，符号表管理，出错管理。这7个部分之间的关联关系如下图所示：

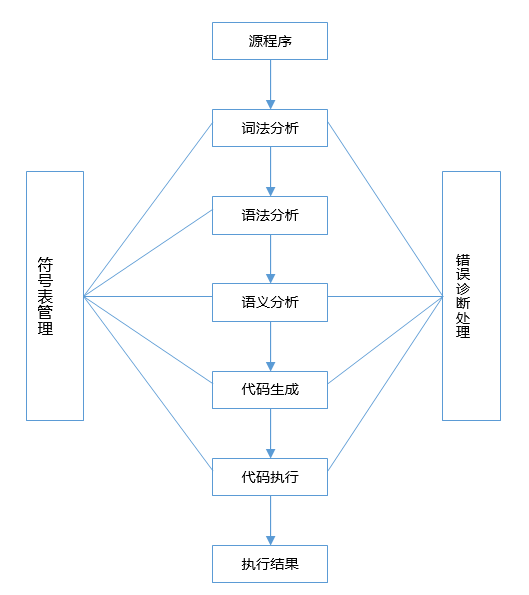


图1 编译程序各部分之间的关联关系

本项目将整个编译过程分为了3个部分：

1、词法分析将Pl/0源代码分为一个个token；

2、语法分析和语义分析同时进行，生成Pcode（包括符号表管理和出错管理）；

3、对Pcode进行解释执行，使其在java虚拟机上能够运行。

整个编译过程可以如下图所示：

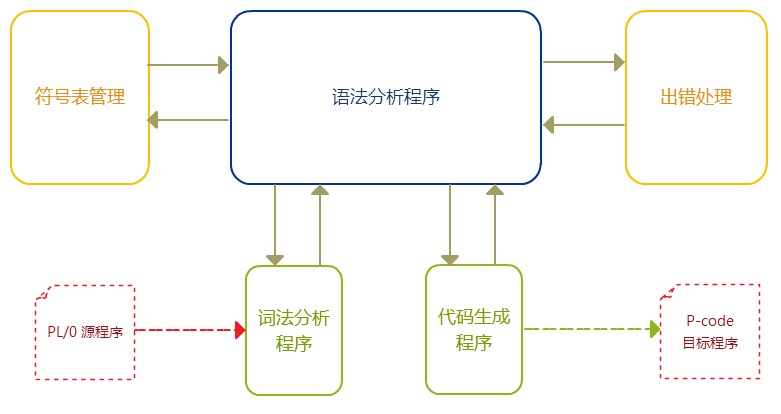


图2 编辑过程流程图

由上图我们可以看到，整个编译程序分为四个小部分（词法分析，出错管理，符号表管理，Pcode生成）和一个综合部分（语法分析）。

下面详细介绍各部分的实现过程。

**实验内容与步骤：**

**1. 词法分析**

词法分析又称词法分析器或者扫描器，是编译程序的基本子程序之一。本项目采用手工方式设计并实现词法分析程序。

1.1 词法分析的功能

扫描源程序，按语言的词法规则识别出各类单词符号（Token），并将有关字符组合成为单词并输出，同时进行词法检查。语言的保留字，标识符，常数和运算符等都是单词的例子。

1.2 Token分类

根据所给的PL/0语言的BNF描述，将PL/0编译系统中所有的字符，字符串的类型按如下表格分类：

表1 PL/0编译系统中所有的字符，字符串的类型

|  |  |  |
| --- | --- | --- |
| 保留字 |  | begin, end, if,then, else, const, procedure,  var,do,while, call,read, write, repeat, until |
| 算数运算符 |  | + ,—，\*，/ |
| 比较运算符 |  | <> , < ,<= , >, >= ,= |
| 赋值符 |  | := , = |
| 标识符 |  | 变量名，过程名，常数名 |
| 常数 |  | 10,25等整数 |
| 界符 |  | ‘，’，‘.’，‘；’，‘（’，‘）’ |
| 其他符号 |  | ：，EOF |

在SymType.java中说明token类型：

**public** **enum** SymType {

***BEG***, ***END***, ***IF***, ***THEN***, ***ELS***, ***CON***, ***PROC***, ***VAR***, ***DO***, ***WHI***, ***CAL***, ***REA***, ***WRI***, ***ODD***, ***REP***, ***UNT***,

//begin, end, if, then, else, const, procedure, var, do, while, call, read, write, odd, repeat, until

***EQU***, ***LES***, ***LESE***, ***LARE***, ***LAR***, ***NEQE***, ***ADD***, ***SUB***, ***MUL***, ***DIV***,

//=, <, <=, >=, >, <>, +, -, \*, /

***SYM***, ***CONST***,

//标识符， 常量

***CEQU***, ***COMMA***, ***SEMIC***, ***POI***, ***LBR***, ***RBR***,

//:=, ',' , ';', '.', '(', ')'

***COL***,

//:

***EOF***;

//end of file

}

1.3 Token结构

在具体实现时，由于出错处理及语法分析的需求，定义如下Token结构：

**public** **class** Token {

**private** SymType st; //token的类别

**private** **int** line; //token所在行，错误处理使用

**private** String value; //token的值，只有标识符和常量有值

}

1.4 Token分析程序的构造

LexAnalysis.java是本项目的词法分析类。其中，doAnalysis()函数对源代码进行了一遍遍历，将源代码中分析成为了一个token数组。

**private** **void** doAnalysis() {

**while** (searchPtr < buffer.length) {

allToken.add(analysis());

}

}

调用analysis函数，生成每一个Token。首先，给出状态图：



图3 词法分析程序的状态转换图

根据状态图，可以写出分析程序analysis()

**private** Token analysis() {

strToken = "";

getChar();

**while** ((ch == ' ' || ch == '\n' || ch == '\t' || ch == '\0') && searchPtr < buffer.length) {

**if** (ch == '\n') {

line++;

}

getChar();

}

**if** (ch == '$' && searchPtr >= buffer.length) { //到达文件末尾

**return** **new** Token(SymType.***EOF***, line, "-1");

}

**if** (isLetter()) { //首位为字母，可能为保留字或者变量名

**while** (isLetter() || isDigit()) {

strToken += ch;

getChar();

}

retract();

**for** (**int** i = 0; i < keyWords.length; i++) {

**if** (strToken.equals(keyWords[i])) { //说明是保留字

**return** **new** Token(SymType.*values*()[i], line, "-");

}

}

//不是保留字，则为标识符，需要保存值

**return** **new** Token(SymType.***SYM***, line, strToken);

} **else** **if** (isDigit()) { //首位为数字，即为整数

**while** (isDigit()) {

strToken += ch;

getChar();

}

retract();

**return** **new** Token(SymType.***CONST***, line, strToken);

} **else** **if** (ch == '=') { //等号

**return** **new** Token(SymType.***EQU***, line, "-");

} **else** **if** (ch == '+') { //加号

**return** **new** Token(SymType.***ADD***, line, "-");

} **else** **if** (ch == '-') { //减号

**return** **new** Token(SymType.***SUB***, line, "-");

} **else** **if** (ch == '\*') { //乘号

**return** **new** Token(SymType.***MUL***, line, "-");

} **else** **if** (ch == '/') { //除号

**return** **new** Token(SymType.***DIV***, line, "-");

} **else** **if** (ch == '<') { //小于或不等于或小于等于

getChar();

**if** (ch == '=') {

**return** **new** Token(SymType.***LESE***, line, "-");

} **else** **if** (ch == '>') {

**return** **new** Token(SymType.***NEQE***, line, "-");

} **else** {

retract();

**return** **new** Token(SymType.***LES***, line, "-");

}

} **else** **if** (ch == '>') { //大于或大于等于

getChar();

**if** (ch == '=') {

**return** **new** Token(SymType.***LARE***, line, "-");

} **else** {

retract();

**return** **new** Token(SymType.***LAR***, line, "-");

}

} **else** **if** (ch == ',') { //逗号

**return** **new** Token(SymType.***COMMA***, line, "-");

} **else** **if** (ch == ';') { //分号

**return** **new** Token(SymType.***SEMIC***, line, "-");

} **else** **if** (ch == '.') { //点

**return** **new** Token(SymType.***POI***, line, "-");

} **else** **if** (ch == '(') { //左括号

**return** **new** Token(SymType.***LBR***, line, "-");

} **else** **if** (ch == ')') { //右括号

**return** **new** Token(SymType.***RBR***, line, "-");

} **else** **if** (ch == ':') { //赋值号

getChar();

**if** (ch == '=') {

**return** **new** Token(SymType.***CEQU***, line, "-");

} **else** {

retract();

**return** **new** Token(SymType.***COL***, line, "-");

}

}

**return** **new** Token(SymType.***EOF***, line, "-");

}

analysis()每次分析出一个Token。对代码进行一遍遍历，即可得到源程序的token数组。

经过上述词法分析后，例语句const a=45; var x,y; 得到的Token表结构为：

|  |  |  |
| --- | --- | --- |
| 符号类型 | 所在行 | 符号值 |
| CON | 1 | - |
| SYM | 1 | a |
| EQU | 1 |  |
| CONST | 1 | 45 |
| SEMIC | 1 | - |
| VAR | 2 | - |
| SYM | 2 | x |
| COMMA | 2 | - |
| SEMIC | 2 | - |

**2. 符号表管理**

符号表是编译过程中编译程序用来记录源程序中的各种名字（即标识符）的特性信息的表格，符号表的每一个登记项将填入名字标识符以及该名字相关联的一些信息，这些信息将全面地反映各个符号的属性及他们在编译过程中的特征。

2.1 符号表结构

**public** **class** PerSymbol {

**private** **int** type; //表示常量1、变量2或过程3

**private** **int** value; //表示常量或变量的值

**private** **int** level; //嵌套层次

**private** **int** address; //相对于所在嵌套过程基地址的地址

**private** **int** size; //表示常量，变量，过程所占的大小

**private** String name; //变量、常量或过程名

}

其中level和address在运行时起到非常大作用。

2.2 符号表管理

为了更好地管理符号表，可以抽象出一些接口供调用，定义在AllSymbol.java中：

表2 各函数名功能表示图

|  |  |  |
| --- | --- | --- |
| 函数名 |  | 功能 |
| enterConst |  | 向符号表中插入常量 |
| enterVar |  | 向符号表中插入变量 |
| enterProc |  | 向符号表中插入过程 |
| isNowExists |  | 在符号表当前层查找变量是否存在 |
| isPreExists |  | 在符号表之前层查找符号是否存在 |
| getSymbol |  | 按名称查找变量 |
| getLevelProc |  | 查找当前层所在的过程 |

2.3 调用示例

在GSAnalysis.java中，用符号表allSymbol记录源程序中的标识符的特性信息，如在子程序conHandle()中如判断当前token值为常量，首先查找在同一嵌套层次中有没有重名的标识符，如果存在同名标识符，则报错；否则在符号表中添加一条常量的定义：

**if** (allSymbol.isNowExists(name, level)) {

errorHandle(15, name);

}

allSymbol.enterConst(name, level, value, address);

**3. 语法语义分析**

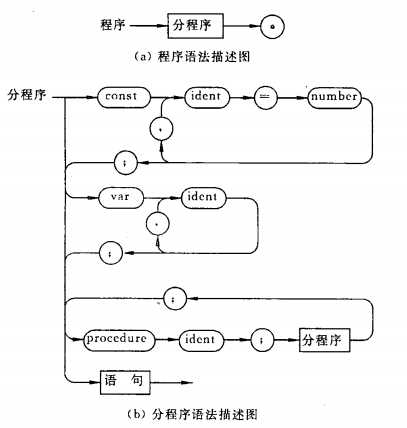
语法分析的任务是识别单词符号序列是否符合给定的语法规则。PL/0编译程序的语法分析采用自顶向下的递归子程序法，即对应每个非终结符语法单元，编一个独立的处理过程（或子程序）。

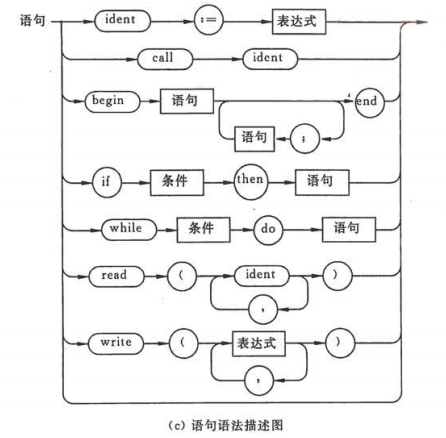
如果一个PL/0语言程序的单词序列在整个语法分析中，都能逐个得到匹配，直到程序结束符“.”，这时所输入的程序是正确的，对于正确的语法分析做相应的语义翻译，生成目标代码。

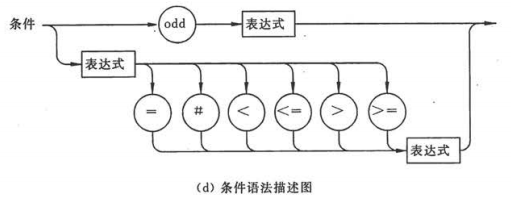
除主程序外，语法语义分析的处理由分程序BLOCK过程完成。

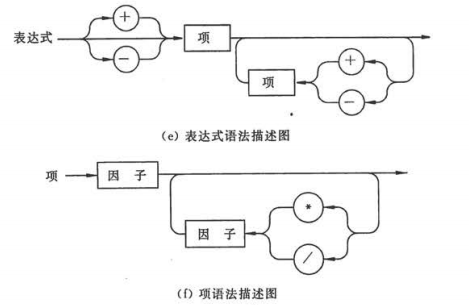
3.1 语法成分分析

用语法图描述语法规则的优点是直观、易读。在图4的语法图中用椭圆和圆圈中的英文字或字符表示终结符，用长方形内的中文字表示非终结符。









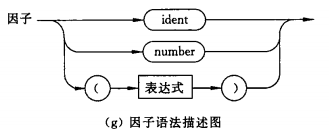


图4 分程序语法描述图

根据语法图，对每一个PL/0中的语法成分进行分析，同时进行符号表管理及Pcode代码生成。各函数定义在GSAnalysis.java中，功能表示如下表：

表3各函数名功能表示图

|  |  |  |
| --- | --- | --- |
| 函数名 |  | 功能 |
| program() |  | <主程序>::=<分程序>. |
| block() |  | <分程序>::=[<常量说明部分>][<变量说明部分>][<过程说明部分>]<语句> |
| conDeclare() |  | <常量说明部分>::=const <常量定义>{,<常量定义>} |
| conHandle() |  | <常量定义>::=<标识符>=<无符号整数> |
| varDeclare() |  | <变量说明部分>::=var<标识符>{,<标识符>} |
| proc() |  | <过程说明部分>::=<过程首部><分程序>{;<过程说明部分>};  <过程首部>::=procedure<标识符>; |
| body() | |  | <复合语句>::=begin<语句>{;<语句>}end |
| statement() |  | <语句>::=<赋值语句> | <条件语句> | <当循环语句> | <过程调用语句> | <  复合语句> | <读语句> | <写语句> | <空> |
| condition() |  | <条件>::=<表达式><关系运算符><表达式> | odd<表达式> |
| expression() |  | <表达式>::=[+|-]<项>{<加法运算符><项>} <加法运算符>::=+|- |
| term() |  | <项>::=<因子>{<乘法运算符><因子>} <乘法运算符>::=\*|/ |
| factor() |  | <因子>::=<标识符> | <无符号整数> | '('<表达式>')' |

3.2 示例

以<常量说明部分>::=const <常量定义>{,<常量定义>}此产生式为列，用伪码描述其子程序如下：

void conDeclare(){

if (当前指向的字符==const){

指向下一个字符;

conHandle();

while(当前指向的字符==’,’ || 当前指向的字符==’标识符’){

if(当前指向的字符==,){

指向下一个字符;

}

else{

error;

}

指向下一个字符;

}

if(当前指向的字符==’;’){

指向下一个字符;

}

else{

error;

}

}

else{

error;

}

}

对应程序代码如下：

**private** **void** conDeclare() {

//<常量说明部分>::=const <常量定义>{,<常量定义>}

**if** (allToken.get(tokenPtr).getSt() == SymType.***CON***) {

tokenPtr++;

conHandle();

**while** (allToken.get(tokenPtr).getSt() == SymType.***COMMA*** || allToken.get(tokenPtr).getSt() == SymType.***SYM***) {

**if** (allToken.get(tokenPtr).getSt() == SymType.***COMMA***) {

tokenPtr++;

} **else** {

errorHandle(23, "");

}

conHandle();

}

**if** (allToken.get(tokenPtr).getSt() == SymType.***SEMIC***) {

tokenPtr++;

} **else** { //缺少；

errorHandle(0, "");

}

} **else** { //缺少const

errorHandle(-1, "");

}

}

**4. Pcode生成**

P-code 语言是一种栈式机的语言。此类栈式机没有累加器和通用寄存器，有一个栈式存储器，有四个控制寄存器（指令寄存器 I，指令地址寄存器 P，栈顶寄存器 T和基址寄存器 B），算术逻辑运算都在栈顶进行。

4.1 Pcode的指令格式为：

|  |  |  |
| --- | --- | --- |
| F | L | A |

指令格式:

F ：操作码

L ：层次差 （标识符引用层减去定义层）

A ：不同的指令含义不同

Pcode结构定义在PerPcode.java中：

**public** **class** PerPcode {

**private** Operator F;

**private** **int** L;

**private** **int** A;

}

对于不同的Pcode指令，各指令含义如下：

表4 P-code 指令的含义

|  |  |
| --- | --- |
| 指令 | 具体含义 |
| LIT 0, a | 取常量a放到数据栈栈顶 |
| OPR 0, a | 执行运算，a表示执行何种运算(+ - \* /) |
| LOD l, a | 取变量放到数据栈栈顶(相对地址为a,层次差为l) |
| STO l, a | 将数据栈栈顶内容存入变量(相对地址为a,层次差为l) |
| CAL l, a | 调用过程(入口指令地址为a,层次差为l) |
| INT 0, a | 数据栈栈顶指针增加a |
| JMP 0, a | 无条件转移到指令地址a |
| JPC 0, a | 条件转移到指令地址a |
| OPR 0 0 | 过程调用结束后,返回调用点并退栈 |
| OPR 0 1 | 栈顶元素取反 |
| OPR 0 2 | 次栈顶与栈顶相加，退两个栈元素，结果值进栈 | |
| OPR 0 3 | 次栈顶减去栈顶，退两个栈元素，结果值进栈 | |
| OPR 0 4 | 次栈顶乘以栈顶，退两个栈元素，结果值进栈 | |
| OPR 0 5 | 次栈顶除以栈顶，退两个栈元素，结果值进栈 | |
| OPR 0 6 | 栈顶元素的奇偶判断，结果值在栈顶 | |
| OPR 0 7 |  | |
| OPR 0 8 | 次栈顶与栈顶是否相等，退两个栈元素，结果值进栈 | |
| OPR 0 9 | 次栈顶与栈顶是否不等，退两个栈元素，结果值进栈 | |
| OPR 0 10 | 次栈顶是否小于栈顶，退两个栈元素，结果值进栈 | |
| OPR 0 11 | 次栈顶是否大于等于栈顶，退两个栈元素，结果值进栈 | |
| OPR 0 12 | 次栈顶是否大于栈顶，退两个栈元素，结果值进栈 | |
| OPR 0 13 | 次栈顶是否小于等于栈顶，退两个栈元素，结果值进栈 | |
| OPR 0 14 | 栈顶值输出至屏幕 | |
| OPR 0 15 | 屏幕输出换行 | |
| OPR 0 16 | 从命令行读入一个输入置于栈顶 | |

根据Pcode指令的相关含义，在语法分析的同时填入Pcode。

4.2 地址回填

在Pcode生成中，地址回填是需要分情况考虑的问题。对于不同的代码，我们采取不同的地址回填策略。

* if-then语句的目标代码生成模式

if <condition> then <statement>

[if]

<condition>

JPC addr1

<statement>

addr1:

* if-then-else语句的目标代码生成模式

if <condition> then <statement>[else]

[if]

<condition>

JPC addr1

<statement>

JMP addr2

addr1: [else]

<statement>

addr2

* while-do语句的目标代码生成模式

while <condition> do <statement>

[while]

addr2: <condition>

JPC addr3

<statement>

JPC addr2

addr3:

* repeat-until语句的目标代码生成模式

repeat <statement> until <condition>

[repeat]

addr4: <statement>

[until]

<condition>

JPC addr4

在statement子过程中分情况讨论当子过程为上述情况时操作码为JPC（条件转移）还是JMP（无条件转移），对于操作码的作用在Interpreter.java中设计实现。

**5. 出错管理**

在编译过程中，既可能出现语法错误，也可能出现语义错误（如变量未声明等）。本项目将错误分为24类，如下表所示：

表5 出错信息表

|  |  |  |
| --- | --- | --- |
| PL/0语言的出错信息表 | | |
| 出错编号 | | 出错原因 | |
| -1 | 常量定义不是const开头，变量定义不是var开头 | |
| 0 | 缺少分号 | |
| 1 | 标识符不合法 | |
| 2 | 不合法的比较符 | |
| 3 | 常量赋值没用= | |
| 4 | 缺少（ | |
| 5 | 缺少） | |
| 6 | 缺少begin | |
| 7 | 缺少end'。 | |
| 8 | 缺少then。 | |
| 9 | 缺少do | |
| 10 | call, write, read语句中，不存在标识符 | |
| 11 | 该标识符不是proc类型 | |
| 12 | read, write语句中，该标识符不是var类型 | |
| 13 | 赋值语句中，该标识符不是var类型 | |
| 14 | 赋值语句中，该标识符不存在 | |
| 15 | 该标识符已存在 | |
| 16 | 调用函数参数错误 | |
| 17 | 缺少. | |
| 18 | 多余代码 | |
| 19 | 缺少until | |
| 20 | 赋值符应为：= | |
| 21 | until前多了； | |
| 22 | 缺少, | |

错误处理函数errorHandle()定义在GSAnalysis.java文件第581行，根据传递来的形参k数值，确定是什么错误。

**6. UI界面设计**

在MyCompiler.java中设计实现PL/0编译程序的UI界面，整个界面分为导航栏，代码区，token表区，符号表区，Pcode区以及控制台。相关实现代码在MyCompiler.java中。

表6 项目各区主要功能介绍

|  |  |  |
| --- | --- | --- |
| 导航栏 |  | 打开，关闭或保存文件；编译和执行代码 |
| 代码区 |  | 展示并编辑代码 |
| Token表区 |  | 展示代码中的token |
| Symbol表区 |  | 展示代码编译过程中生成的symbol表 |
| Pcode表区 |  | 展示代码生成的所有Pcode |
| 控制台 |  | 输出编译信息 |

6.1 功能演示

打开一段代码到代码区，选择编译。

编译成过则会显示成功信息并展示相应的token表，symbol表及Pcode表。

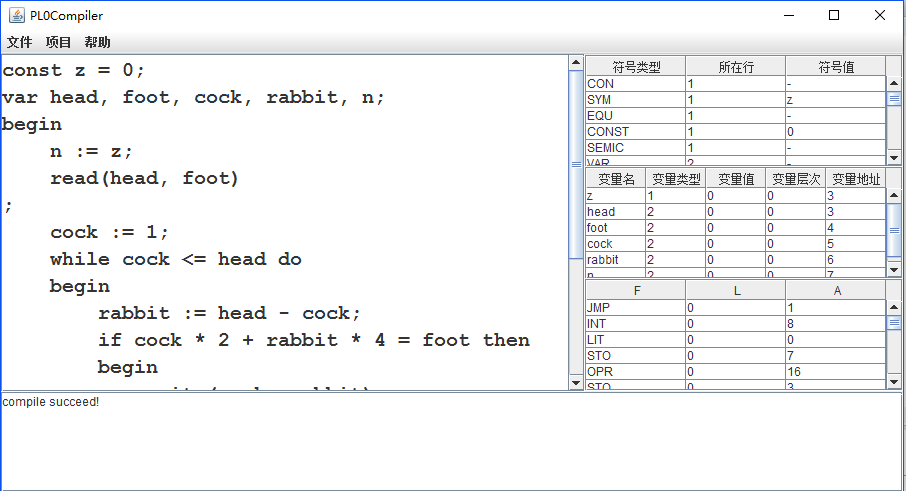


图5 运行成功示意图

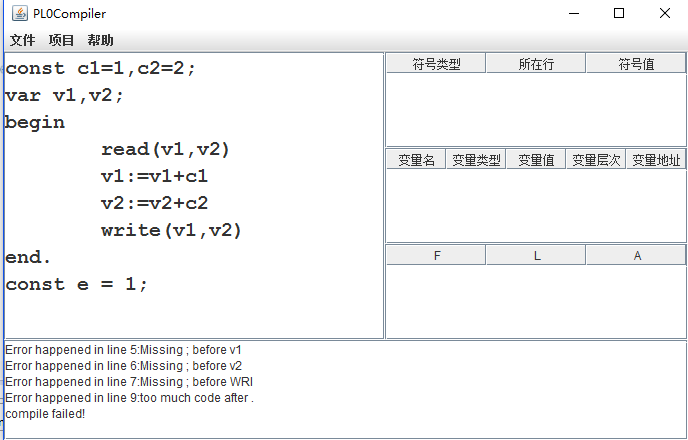


图6 编译失败示意图

在“文件”栏下可以选择“打开”、“保存”、“关闭”：

openItem.addActionListener(**new** ActionListener() {//菜单条目监听：打开

**public** **void** actionPerformed(ActionEvent e) {

open.setVisible(**true**);

String dirPath = open.getDirectory();

String fileName = open.getFile();

**if** (dirPath == **null** || fileName == **null**) {

**return**;

}

file = **new** File(dirPath, fileName);

jTextArea.setText("");//打开文件之前清空文本区域

**try** {

BufferedReader br= **new** BufferedReader(**new** FileReader(file));

String line = **null**;

**while** ((line = br.readLine()) != **null**) {

//将给定文本追加到文档结尾。如果模型为 null 或者字符串为 null 或空，则不执行任何操作。

//虽然大多数 Swing 方法不是线程安全的，但此方法是线程安全的。

jTextArea.append(line + "\r\n");

}

}

**catch** (IOException ex) {

**throw** **new** RuntimeException("读取失败！");

}

}

});

saveItem.addActionListener(**new** ActionListener() {//菜单条目监听：保存

**public** **void** actionPerformed(ActionEvent e) {

**if** (file == **null**) {

newFile();

}

saveFile();

}

});

closeItem.addActionListener(**new** ActionListener() {

@Override

**public** **void** actionPerformed(ActionEvent e) {

System.*exit*(0);

}

});

在“项目”下可以选择“编译”、“运行”：

compileItem.addActionListener(**new** ActionListener() {

**public** **void** actionPerformed(ActionEvent e) {

compile();

}

});

runItem.addActionListener(**new** ActionListener() {

**public** **void** actionPerformed(ActionEvent e) {

**if** (success) {

run();

}

}

});

在“帮助”栏下可以查看相关信息“关于”：

aboutItem.addActionListener(**new** ActionListener() {

@Override

**public** **void** actionPerformed(ActionEvent e) {

JOptionPane.*showMessageDialog*(**null**, "PL0Compiler\n"

+ "made by yuqingjin\nyuqingjin2017@163.com");

}

});

设置token、symbol、pcode表格：

//设置token表格

tokenTable = **new** JTable(tokenTableModel);

tokenTable.setPreferredScrollableViewportSize(**new** Dimension(300, 100));

tokenTable.setFillsViewportHeight(**true**);

tokenJScrollPane = **new** JScrollPane(tokenTable);

//设置symbbol表格

symbolTable = **new** JTable(symbolTableModel);

symbolTable.setPreferredScrollableViewportSize(**new** Dimension(300, 100));

symbolTable.setFillsViewportHeight(**true**);

symbolJScrollPane = **new** JScrollPane(symbolTable);

//设置pcode表格

pcodeTable = **new** JTable(pcodeTableModel);

pcodeTable.setPreferredScrollableViewportSize(**new** Dimension(300, 100));

pcodeTable.setFillsViewportHeight(**true**);

pcodeJScrollPane = **new** JScrollPane(pcodeTable);

tablePanel = **new** JPanel();

tablePanel.setLayout( **new** GridLayout (0, 1));

tablePanel.add(tokenJScrollPane);

tablePanel.add(symbolJScrollPane);

tablePanel.add(pcodeJScrollPane);

//出错信息

errorMessage = **new** JTextArea();

errorPane = **new** JScrollPane(errorMessage);

errorPane.setPreferredSize(**new** Dimension(700, 100));

//放置菜单项及输入框

frame.add(menuBar, BorderLayout.***NORTH***);

frame.add(jScrollPane, BorderLayout.***CENTER***);

frame.add(tablePanel, BorderLayout.***EAST***);

frame.add(errorPane, BorderLayout.***SOUTH***);

**实验数据及处理：**

本项目提供5个测试用例。

1）text1.txt——鸡兔同笼问题

const z = 0;

var head, foot, cock, rabbit, n;

begin

n := z;

read(head, foot)

;

cock := 1;

while cock <= head do

begin

rabbit := head - cock;

if cock \* 2 + rabbit \* 4 = foot then

begin

write(cock, rabbit);

n := n + 1

end;

cock := cock + 1

end;

if n = 0 then write(0, 0)

end.

2）text2.txt——求最大公约数和最小公倍数

const a = 45, b = 27;

var x, y, g, m;

procedure swap;

var temp;

begin

temp := x;

x := y;

y := temp

end;

procedure mod;

x := x - x / y \* y;

begin

x := a;

y := b;

call mod;

while x <> 0 do

begin

call swap;

call mod

end;

g := y;

m := a \* b / g;

write(g, m)

end.

3）text3.txt——找素数

const true = 1, false = 0;

var x, y, m, n, pf;

procedure prime;

var i, f;

procedure mod;

x := x - x / y \* y;

begin

f := true;

i := 3;

while i < m do

begin

x := m;

y := i;

call mod;

if x = 0 then f := false;

i := i + 2

end;

if f = true then

begin

write(m);

pf := true

end

end;

begin

pf := false;

read(n);

while n >= 2 do

begin

write(2);

if n = 2 then pf := true;

m := 3;

while m <= n do

begin

call prime;

m := m + 2

end;

read(n)

end;

if pf = false then write(0)

end.

4）text4.txt——存在语法错误

const c1=1,c2=2;

var v1,v2;

begin

read(v1,v2)

v1:=v1+c1

v2:=v2+c2

write(v1,v2)

end.

const e = 1;

5）text5.txt——存在语义错误

const c1=1,c2=2;

var v1,v2;

begin

read(v1,v2);

v1:=v1+c1;

v3:=v2+c2;

write(v1,v2)

end.

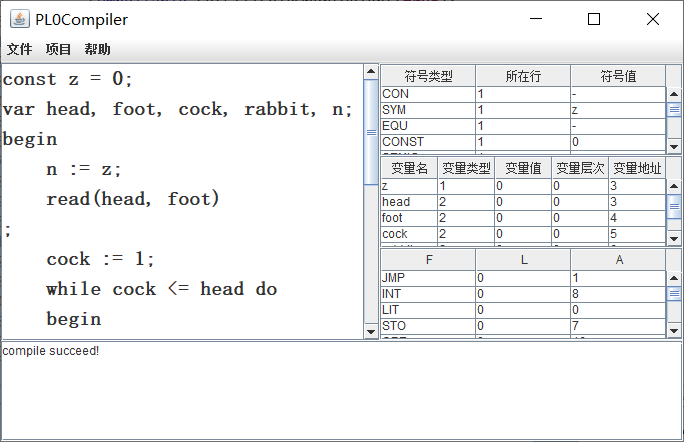
**实验结果与分析：**

**（1）实验结果及分析**

运行方法：拷贝源码src文件下的所有代码，在eclipse中新建Java项目（JRE：1.8），将代码粘贴到src文件夹下，直接运行即可。

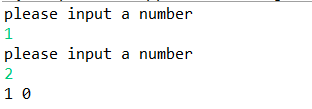
1）text1.txt——鸡兔同笼问题

①点击“编译”



右侧依次为token、symbol、pcode表格，经检查正确，在下方提示编译成功。

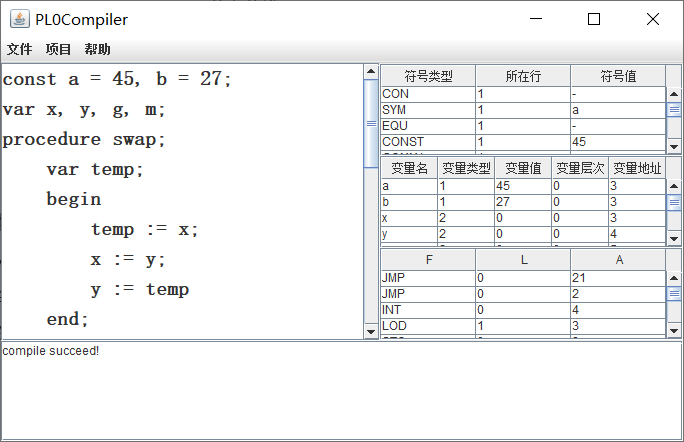
②点击“运行”



根据代码可知，第一个数字为头总数，第二个数字为脚总数。运行结果为鸡1只，兔0只，可知运行结果正确。

2）text2.txt

①点击“编译”



右侧依次为token、symbol、pcode表格，经检查正确，在下方提示编译成功。

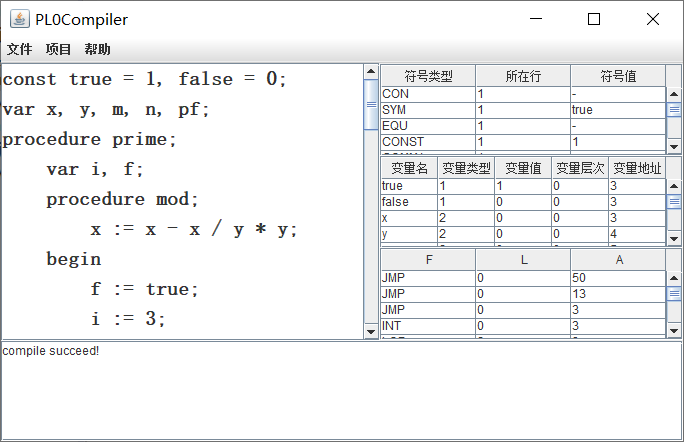
②点击“运行”



根据代码可知，输入a = 45，b = 27，得到的最大公因数为9，最小公倍数为135，可知运行结果正确。

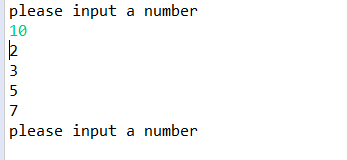
3）text3.txt

①点击“编译”



右侧依次为token、symbol、pcode表格，经检查正确，在下方提示编译成功。

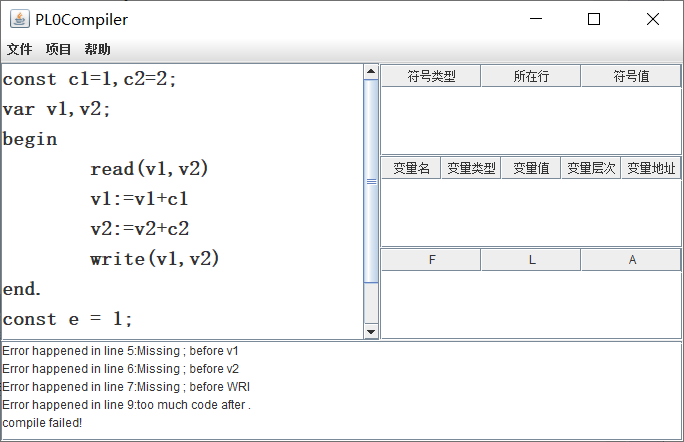
②点击“运行”



根据代码可知，输入10，得到10以内的素数为2，3，5，7，可知运行结果正确。

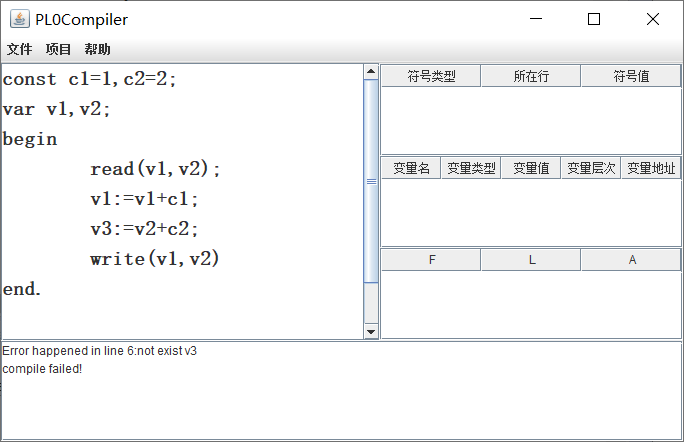
4）text4.txt

①编译



在该代码的第4，5，6行最后没有加”;”，存在语法错误，成功报错，在第8行已经是代码末尾”end.”后仍有代码，成功报错。

5）text5.txt



在该代码第6行，变量v3没有提前声明，存在语义错误，成功报错。

**（2）实验总结**

通过本次实验实现了对PL/0源代码的词法分析、语法分析、语义分析及Pcode代码的生成，还对Pcode代码进行了解释，使其能在Java虚拟机上运行，深入了解了编译程序的编译过程。从PL/0文法的EBNF表示入手，根据PL/0编译程序的特点，通过递归下降子程序方法，设计实现了PL/0编译程序的UI界面，对于课程学习起到了很好的巩固作用，提高了动手实践能力。

教师评审

|  |  |
| --- | --- |
| 教师评语 | 实验成绩 |
| 签名：  日期： |  |

**附录**

**MyCompiler.java**

import java.awt.BorderLayout;

import java.awt.Dimension;

import java.awt.FileDialog;

import java.awt.Font;

import java.awt.GridLayout;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.File;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;

import javax.swing.JFileChooser;

import javax.swing.JFrame;

import javax.swing.JMenu;

import javax.swing.JMenuBar;

import javax.swing.JMenuItem;

import javax.swing.JOptionPane;

import javax.swing.JPanel;

import javax.swing.JScrollPane;

import javax.swing.JTable;

import javax.swing.JTextArea;

import javax.swing.table.DefaultTableModel;

import javax.swing.table.TableModel;

public class MyCompiler extends JFrame{

private JMenuBar menuBar;

private JMenu fileMenu;

private JMenu projectMenu;

private JMenu helpMenu;

private JTextArea jTextArea;

private JScrollPane jScrollPane;

private JMenuItem openItem, closeItem, saveItem,aboutItem;

private JMenuItem compileItem, runItem;

private FileDialog open,save;

private File file;

private JPanel tablePanel; //放置所有表格

//token表格

private JScrollPane tokenJScrollPane;

private JTable tokenTable;

String[] tokenColumnNames = {"符号类型","所在行", "符号值"};

private TableModel tokenTableModel = new DefaultTableModel(tokenColumnNames, 0);

//symbol表格

private JScrollPane symbolJScrollPane;

private JTable symbolTable;

String[] symbolColumnNames = {"变量名", "变量类型", "变量值", "变量层次", "变量地址"};

private TableModel symbolTableModel = new DefaultTableModel(symbolColumnNames, 0);

//pcode表格

private JScrollPane pcodeJScrollPane;

private JTable pcodeTable;

String[] pcodeColumnNames = {"F", "L", "A"};

private TableModel pcodeTableModel = new DefaultTableModel(pcodeColumnNames, 0);

private JTextArea errorMessage;

private JScrollPane errorPane;

private GSAnalysis gsa;

private List<Token> allToken;

private List<PerSymbol> allSymbol;

private List<PerPcode> allPcode;

private List<String> errors;

private String consoleMessage;

private int readNum = 0;

private List<String> output;

private boolean success = false;

public MyCompiler() {

init();

}

private void init() {

JFrame frame = new JFrame("PL0Compiler");

frame.setBounds(300, 300, 700, 450);

frame.setDefaultCloseOperation(EXIT\_ON\_CLOSE);

menuBar = new JMenuBar();//菜单栏

fileMenu = new JMenu("文件");

projectMenu = new JMenu("项目");

helpMenu = new JMenu("帮助");

jTextArea = new JTextArea(10, 40);

jTextArea.setFont(new Font("Monospaced",1,20));

jTextArea.setLineWrap(true);//到达指定宽度则换行

//应当首先利用构造函数指定JScrollPane的控制对象，此处为JTextArea，然后再添加JScrollPane

//添加进面板

jScrollPane = new JScrollPane(jTextArea);

//设置滚动条自动出现

jScrollPane.setHorizontalScrollBarPolicy( JScrollPane.HORIZONTAL\_SCROLLBAR\_AS\_NEEDED);

jScrollPane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL\_SCROLLBAR\_AS\_NEEDED);

jScrollPane.setViewportView(jTextArea);

openItem = new JMenuItem("打开");

saveItem = new JMenuItem("保存");

closeItem = new JMenuItem("关闭");

aboutItem = new JMenuItem("关于");

compileItem = new JMenuItem("编译");

runItem = new JMenuItem("运行");

//添加两个选项卡到JMenu

//添加字菜单项到菜单项

menuBar.add(fileMenu);

menuBar.add(projectMenu);

menuBar.add(helpMenu);

fileMenu.add(openItem);

fileMenu.add(saveItem);

fileMenu.add(closeItem);

projectMenu.add(compileItem);

projectMenu.add(runItem);

helpMenu.add(aboutItem);

//设置token表格

tokenTable = new JTable(tokenTableModel);

tokenTable.setPreferredScrollableViewportSize(new Dimension(300, 100));

tokenTable.setFillsViewportHeight(true);

tokenJScrollPane = new JScrollPane(tokenTable);

//设置symbol表格

symbolTable = new JTable(symbolTableModel);

symbolTable.setPreferredScrollableViewportSize(new Dimension(300, 100));

symbolTable.setFillsViewportHeight(true);

symbolJScrollPane = new JScrollPane(symbolTable);

//设置pcode表格

pcodeTable = new JTable(pcodeTableModel);

pcodeTable.setPreferredScrollableViewportSize(new Dimension(300, 100));

pcodeTable.setFillsViewportHeight(true);

pcodeJScrollPane = new JScrollPane(pcodeTable);

tablePanel = new JPanel();

tablePanel.setLayout( new GridLayout (0, 1));

tablePanel.add(tokenJScrollPane);

tablePanel.add(symbolJScrollPane);

tablePanel.add(pcodeJScrollPane);

//出错信息

errorMessage = new JTextArea();

errorPane = new JScrollPane(errorMessage);

errorPane.setPreferredSize(new Dimension(700, 100));

//放置菜单项及输入框

frame.add(menuBar, BorderLayout.NORTH);

frame.add(jScrollPane, BorderLayout.CENTER);

frame.add(tablePanel, BorderLayout.EAST);

frame.add(errorPane, BorderLayout.SOUTH);

open = new FileDialog(frame,"打开文档",FileDialog.LOAD);

save = new FileDialog(frame,"保存文档",FileDialog.SAVE);

Event();

frame.setVisible(true);

}

private void Event() {

closeItem.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

System.exit(0);

}

});

aboutItem.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

JOptionPane.showMessageDialog(null, "PL0Compiler\n"

+ "made by yuqingjin\nyuqingjin2017@163.com");

}

});

openItem.addActionListener(new ActionListener() {//菜单条目监听：打开

public void actionPerformed(ActionEvent e) {

open.setVisible(true);

String dirPath = open.getDirectory();

String fileName = open.getFile();

if (dirPath == null || fileName == null) {

return;

}

file = new File(dirPath, fileName);

jTextArea.setText("");//打开文件之前清空文本区域

try {

BufferedReader br = new BufferedReader(new FileReader(file));

String line = null;

while ((line = br.readLine()) != null) {

//将给定文本追加到文档结尾。如果模型为 null 或者字符串为 null 或空，则不执行任何操作。

//虽然大多数 Swing 方法不是线程安全的，但此方法是线程安全的。

jTextArea.append(line + "\r\n");

}

}

catch (IOException ex) {

throw new RuntimeException("读取失败！");

}

}

});

saveItem.addActionListener(new ActionListener() {//菜单条目监听：保存

public void actionPerformed(ActionEvent e) {

if (file == null) {

newFile();

}

saveFile();

}

});

compileItem.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

compile();

}

});

runItem.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

if (success) {

//该函数还没有做好

run();

}

}

});

}

//运行Pcode，目前不知道怎么传递输入，尤其是输入写在循环中时。

private void run() {

gsa.interpreter();

/\*\*\*\*\*\*\*\*

List<Integer> input = new ArrayList<Integer>();

String s = errorMessage.getText();

String[] sp = s.split("\n");

for (int i = 2; i < sp.length; i++) {

//System.out.println(sp[i]);

input.add(Integer.parseInt(sp[i]));

}

output = gsa.interpreter(input);

for (int i = 0; i < output.size(); i++) {

consoleMessage += output.get(i);

}

errorMessage.setText(consoleMessage);

consoleMessage = "";

\*\*\*\*\*\*\*\*\*\*\*\*/

}

//编译

private void compile() {

if (file == null) {

JOptionPane.showMessageDialog(null, "请先保存文件");

newFile();

}

saveFile();

gsa = new GSAnalysis(file);

clean();

if (success = gsa.compile()) {

displayAllToken();

displayAllSymbol();

displayAllPcode();

consoleMessage += "compile succeed!\n";

//consoleMessage += "请输入" + readNum + "个数，每行一个\n";

errorMessage.setText(consoleMessage);

} else {

displayErrorMessage();

consoleMessage += "compile failed!";

errorMessage.setText(consoleMessage);

}

}

//编译前清理一些表格和字符串

private void clean() {

flushTable(tokenTable);

flushTable(symbolTable);

flushTable(pcodeTable);

errorMessage.setText("");

consoleMessage = "";

success = false;

}

private void displayErrorMessage() {

consoleMessage = "";

errors = gsa.getErrorMessage();

for (int i = 0; i < errors.size(); i++) {

consoleMessage += errors.get(i) + "\n";

}

}

private void displayAllToken() {

DefaultTableModel model = (DefaultTableModel)tokenTable.getModel();

allToken = gsa.getAllToken();

for (int i = 0; i < allToken.size(); i++) {

Token token = allToken.get(i);

Object[] rowValues = {token.getSt(), token.getLine(), token.getValue()};

model.addRow(rowValues);

}

}

private void displayAllSymbol() {

DefaultTableModel model = (DefaultTableModel)symbolTable.getModel();

allSymbol = gsa.getAllSymbol();

for (int i = 0; i < allSymbol.size(); i++) {

PerSymbol symbol = allSymbol.get(i);

Object[] rowValues = {symbol.getName(), symbol.getType(), symbol.getValue(), symbol.getLevel(), symbol.getAddress()};

model.addRow(rowValues);

}

}

private void displayAllPcode() {

DefaultTableModel model = (DefaultTableModel)pcodeTable.getModel();

allPcode = gsa.getAllPcode();

readNum = 0;

for (int i = 0; i < allPcode.size(); i++) {

PerPcode pcode = allPcode.get(i);

if (pcode.getF() == Operator.OPR && pcode.getA() == 16) {

readNum++;

}

Object[] rowValues = {pcode.getF(), pcode.getL(), pcode.getA()};

model.addRow(rowValues);

}

}

private void flushTable(JTable table) {

((DefaultTableModel) table.getModel()).getDataVector().clear(); //清除表格数据

((DefaultTableModel) table.getModel()).fireTableDataChanged();//通知模型更新

table.updateUI();//刷新表格

}

//新建一个文件

private void newFile() {

if (file == null) {

save.setVisible(true);

String dirPath = save.getDirectory();

String fileName = save.getFile();

if(dirPath == null || fileName == null) {

return;

}

file = new File(dirPath, fileName);

}

}

//保存文件

private void saveFile() {

try {

BufferedWriter bw = new BufferedWriter(new FileWriter(file));

String text = jTextArea.getText();

bw.write(text);

bw.close();

} catch (IOException ex) {

throw new RuntimeException();

}

}

public static void main(String[] args) {

new MyCompiler();

}

}

**AllPcode.java**

import java.util.ArrayList;

import java.util.List;

/\*\*

\* 保存所有Pcode指令

\*/

public class AllPcode {

List<PerPcode> allPcode;

/\*\*

\* 代码的具体形式：

\* FLA

\* 其中：F段代表伪操作码

\* L段代表调用层与说明层的层差值

\* A段代表位移量（相对地址）

\* 进一步说明：

\* INT：为被调用的过程（包括主过程）在运行栈S中开辟数据区，这时A段为所需数据单元个数（包括三个连接数据）；L段恒为0。

\* CAL：调用过程，这时A段为被调用过程的过程体（过程体之前一条指令）在目标程序区的入口地址。

\* LIT：将常量送到运行栈S的栈顶，这时A段为常量值。

\* LOD：将变量送到运行栈S的栈顶，这时A段为变量所在说明层中的相对位置。

\* STO：将运行栈S的栈顶内容送入某个变量单元中，A段为变量所在说明层中的相对位置。

\* JMP：无条件转移，这时A段为转向地址（目标程序）。

\* JPC：条件转移，当运行栈S的栈顶的布尔值为假（0）时，则转向A段所指目标程序地址；否则顺序执行。

\* OPR：关系或算术运算，A段指明具体运算，例如A=2代表算术运算“＋”；A＝12代表关系运算“>”等等。运算对象取自运行栈S的栈顶及次栈顶。

\*

\* OPR 0 0 过程调用结束后,返回调用点并退栈

\* OPR 0 1 栈顶元素取反

\* OPR 0 2 次栈顶与栈顶相加，退两个栈元素，结果值进栈

\* OPR 0 3 次栈顶减去栈顶，退两个栈元素，结果值进栈

\* OPR 0 4 次栈顶乘以栈顶，退两个栈元素，结果值进栈

\* OPR 0 5 次栈顶除以栈顶，退两个栈元素，结果值进栈

\* OPR 0 6 栈顶元素的奇偶判断，结果值在栈顶

\* OPR 0 7

\* OPR 0 8 次栈顶与栈顶是否相等，退两个栈元素，结果值进栈

\* OPR 0 9 次栈顶与栈顶是否不等，退两个栈元素，结果值进栈

\* OPR 0 10 次栈顶是否小于栈顶，退两个栈元素，结果值进栈

\* OPR 0 11 次栈顶是否大于等于栈顶，退两个栈元素，结果值进栈

\* OPR 0 12 次栈顶是否大于栈顶，退两个栈元素，结果值进栈

\* OPR 0 13 次栈顶是否小于等于栈顶，退两个栈元素，结果值进栈

\* OPR 0 14 栈顶值输出至屏幕

\* OPR 0 15 屏幕输出换行

\* OPR 0 16 从命令行读入一个输入置于栈顶

\*/

public AllPcode() {

allPcode = new ArrayList<PerPcode>();

}

public List<PerPcode> getAllPcode() {

return allPcode;

}

public int getPcodePtr() {

return allPcode.size();

}

public void gen(PerPcode pcode) {

allPcode.add(pcode);

}

public void gen(Operator L, int F, int A) {

allPcode.add(new PerPcode(L, F, A));

}

}

**AllSymbol.java**

import java.util.ArrayList;

import java.util.List;

/\*\*

\* 符号表

\*/

public class AllSymbol {

List<PerSymbol> allSymbol;

private int con=1; //常量类型用1表示

private int var=2; //变量类型用2表示

private int proc=3; //过程类型用3表示

private int ptr = 0;

public AllSymbol() {

allSymbol = new ArrayList<PerSymbol>();

}

//向符号表中插入常量

public void enterConst(String name, int level, int value, int address) {

allSymbol.add(new PerSymbol(con, value, level, address, 0, name));

ptr++;

}

//向符号表中插入变量

public void enterVar(String name, int level, int address) {

allSymbol.add(new PerSymbol(var, level, address, 0, name));

ptr++;

}

//向符号表中插入过程

public void enterProc(String name, int level, int address) {

allSymbol.add(new PerSymbol(proc, level, address, 0, name));

ptr++;

}

//在符号表当前层查找变量是否存在

public boolean isNowExists(String name, int level) {

for (int i = 0; i < allSymbol.size(); i++) {

if (allSymbol.get(i).getName().equals(name) && allSymbol.get(i).getLevel() == level) {

return true;

}

}

return false;

}

//在符号表之前层查找符号是否存在

public boolean isPreExists(String name, int level) {

for (int i = 0; i < allSymbol.size(); i++) {

if (allSymbol.get(i).getName().equals(name) && allSymbol.get(i).getLevel() <= level) {

return true;

}

}

return false;

}

//按名称查找变量

public PerSymbol getSymbol(String name) {

for (int i = allSymbol.size() - 1; i >= 0; i--) {

if (allSymbol.get(i).getName().equals(name)) {

return allSymbol.get(i);

}

}

return null;

}

//查找当前层所在的过程

public int getLevelProc(int level) {

for (int i = allSymbol.size() - 1; i >= 0; i--) {

if (allSymbol.get(i).getType() == proc) {

return i;

}

}

return -1;

}

public List<PerSymbol> getAllSymbol() {

return allSymbol;

}

public void setPtr(int \_ptr) {

ptr = \_ptr;

}

public int getPtr() {

return ptr;

}

public int getLength() {

return allSymbol.size();

}

public int getCon() {

return con;

}

public int getVar() {

return var;

}

public int getProc() {

return proc;

}

}

**GSAnalysis.java**

import java.io.File;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* 语法分析+语义分析（生成符号表和Pcode）

\* 错误处理

\*/

public class GSAnalysis {

private LexAnalysis lex;

private List<Token> allToken; //保存词法分析结果

private AllPcode allPcode; //保存生成的Pcode

private AllSymbol allSymbol; //符号表管理

private List<String> errorMessage; //保存错误信息

private boolean errorHappen = false; //记录编译过程中是否发生错误

private int tokenPtr = 0; //指向当前token的指针

private int level = 0;

private int address = 0;

private int addIncrement = 1;

GSAnalysis(File file) {

lex = new LexAnalysis(file);

allToken = lex.getAllToken();

allPcode = new AllPcode();

allSymbol = new AllSymbol();

errorMessage = new ArrayList<String>();

}

public boolean compile() {

program();

return (!errorHappen);

}

private void program() {

//<主程序>::=<分程序>.

block();

if (allToken.get(tokenPtr).getSt() == SymType.POI) {

tokenPtr++;

if (allToken.get(tokenPtr).getSt() != SymType.EOF) {

errorHandle(18, "");

}

} else {

errorHandle(17, "");

}

}

private void block() {

//<分程序>::=[<常量说明部分>][<变量说明部分>][<过程说明部分>]<语句>

int address\_cp = address; //保存之前的address值

//初始化本层的相关变量

int start = allSymbol.getPtr(); //本层变量声明的初始位置

int pos = 0; //本层过程声明在符号表中的位置

address = 3; //默认已3开始，前几位存放一些跳转关键变量，如原来的base（基地址），pc（程序计数器）等

if (start > 0) {

pos = allSymbol.getLevelProc(level);

}

//设置跳转指令，跳过声明部分，后面回填

int tmpPcodePtr = allPcode.getPcodePtr();

allPcode.gen(Operator.JMP, 0, 0);

if (allToken.get(tokenPtr).getSt() == SymType.CON) {

conDeclare();

}

if (allToken.get(tokenPtr).getSt() == SymType.VAR) {

varDeclare();

}

if (allToken.get(tokenPtr).getSt() == SymType.PROC) {

proc();

}

allPcode.getAllPcode().get(tmpPcodePtr).setA(allPcode.getPcodePtr()); //回填跳转地址

allPcode.gen(Operator.INT, 0, address); //申请空间

if (start != 0) {

//如果不是主函数，则需要在符号表中的value填入该过程在Pcode代码中的起始位置

allSymbol.getAllSymbol().get(pos).setValue(allPcode.getPcodePtr() - 1 - allSymbol.getAllSymbol().get(pos).getSize());

}

statement();

allPcode.gen(Operator.OPR, 0, 0); //过程结束

address = address\_cp;

}

private void conDeclare() {

//<常量说明部分>::=const <常量定义>{,<常量定义>}

if (allToken.get(tokenPtr).getSt() == SymType.CON) {

tokenPtr++;

conHandle();

while (allToken.get(tokenPtr).getSt() == SymType.COMMA || allToken.get(tokenPtr).getSt() == SymType.SYM) {

if (allToken.get(tokenPtr).getSt() == SymType.COMMA) {

tokenPtr++;

} else {

errorHandle(23, "");

}

conHandle();

}

if (allToken.get(tokenPtr).getSt() == SymType.SEMIC) {

tokenPtr++;

} else { //缺少；

errorHandle(0, "");

}

} else { //缺少const

errorHandle(-1, "");

}

}

private void conHandle() {

//<常量定义>::=<标识符>=<无符号整数>

String name;

int value;

if (allToken.get(tokenPtr).getSt() == SymType.SYM) {

name = allToken.get(tokenPtr).getValue();

tokenPtr++;

if (allToken.get(tokenPtr).getSt() == SymType.EQU || allToken.get(tokenPtr).getSt() == SymType.CEQU) {

if (allToken.get(tokenPtr).getSt() == SymType.CEQU) {

errorHandle(3, "");

}

tokenPtr++;

if (allToken.get(tokenPtr).getSt() == SymType.CONST) {

value = Integer.parseInt(allToken.get(tokenPtr).getValue());

if (allSymbol.isNowExists(name, level)) {

errorHandle(15, name);

}

allSymbol.enterConst(name, level, value, address);

//address += addIncrement;

tokenPtr++;

}

} else { //赋值没用=

errorHandle(3, "");

}

} else { //标识符不合法

errorHandle(1, "");

}

}

private void varDeclare() {

//<变量说明部分>::=var<标识符>{,<标识符>}

String name;

int value;

if (allToken.get(tokenPtr).getSt() == SymType.VAR) {

tokenPtr++;

if (allToken.get(tokenPtr).getSt() == SymType.SYM) {

name = allToken.get(tokenPtr).getValue();

if (allSymbol.isNowExists(name, address)) {

errorHandle(15, name);

}

allSymbol.enterVar(name, level, address);

address += addIncrement;

tokenPtr++;

while (allToken.get(tokenPtr).getSt() == SymType.COMMA || allToken.get(tokenPtr).getSt() == SymType.SYM) {

if (allToken.get(tokenPtr).getSt() == SymType.COMMA) {

tokenPtr++;

} else {

errorHandle(23, "");

}

if (allToken.get(tokenPtr).getSt() == SymType.SYM) {

name = allToken.get(tokenPtr).getValue();

if (allSymbol.isNowExists(name, address)) {

errorHandle(15, name);

}

allSymbol.enterVar(name, level, address);

address += addIncrement;

tokenPtr++;

} else { //非法标识符

errorHandle(1, "");

return;

}

}

if (allToken.get(tokenPtr).getSt() != SymType.SEMIC) { //缺少；

errorHandle(0, "");

return;

} else {

tokenPtr++;

}

} else { //非法标识符

errorHandle(1, "");

return;

}

} else { //缺少var

errorHandle(-1, "");

return;

}

}

private void proc() {

//<过程说明部分>::=<过程首部><分程序>{;<过程说明部分>};

//<过程首部>::=procedure<标识符>;

if (allToken.get(tokenPtr).getSt() == SymType.PROC) {

tokenPtr++;

int count = 0; //记录参数个数

int pos; //记录该过程在符号表中的位置

if (allToken.get(tokenPtr).getSt() == SymType.SYM) {

String name = allToken.get(tokenPtr).getValue();

if (allSymbol.isNowExists(name, level)) {

errorHandle(15, name);

}

pos = allSymbol.getPtr();

allSymbol.enterProc(name, level, address);

address += addIncrement;

level++;

tokenPtr++;

if (allToken.get(tokenPtr).getSt() == SymType.SEMIC) {

tokenPtr++;

} else {

errorHandle(0, "");

}

block();

while (allToken.get(tokenPtr).getSt() == SymType.SEMIC || allToken.get(tokenPtr).getSt() == SymType.PROC) {

if (allToken.get(tokenPtr).getSt() == SymType.SEMIC) {

tokenPtr++;

} else {

errorHandle(0, "");

}

level--;

proc();

}

} else {

errorHandle(-1, "");

return;

}

}

}

private void body() {

//<复合语句>::=begin<语句>{;<语句>}end

if (allToken.get(tokenPtr).getSt() == SymType.BEG) {

tokenPtr++;

statement();

while (allToken.get(tokenPtr).getSt() == SymType.SEMIC || isHeadOfStatement()) {

if (allToken.get(tokenPtr).getSt() == SymType.SEMIC) {

tokenPtr++;

} else {

if (allToken.get(tokenPtr).getSt() != SymType.END) {

errorHandle(0, "");

}

}

if (allToken.get(tokenPtr).getSt() == SymType.END) {

errorHandle(21, "");

break;

}

statement();

}

if (allToken.get(tokenPtr).getSt() == SymType.END) {

tokenPtr++;

} else { //缺少end

errorHandle(7, "");

return;

}

} else { //缺少begin

errorHandle(6, "");

return;

}

}

private void statement() {

//<语句>::=<赋值语句> | <条件语句> | <当循环语句> | <过程调用语句> | <复合语句> | <读语句> | <写语句> | <空>

if (allToken.get(tokenPtr).getSt() == SymType.IF) {

//<条件语句>::=if<条件>then<语句>else<语句>

tokenPtr++;

condition();

if (allToken.get(tokenPtr).getSt() == SymType.THEN) {

int pos1 = allPcode.getPcodePtr();

allPcode.gen(Operator.JPC, 0, 0);

tokenPtr++;

statement();

int pos2 = allPcode.getPcodePtr();

allPcode.gen(Operator.JMP, 0, 0);

allPcode.getAllPcode().get(pos1).setA(allPcode.getPcodePtr());

allPcode.getAllPcode().get(pos2).setA(allPcode.getPcodePtr());

if (allToken.get(tokenPtr).getSt() == SymType.ELS) {

tokenPtr++;

statement();

allPcode.getAllPcode().get(pos2).setA(allPcode.getPcodePtr());

}

} else {

errorHandle(8, "");

return;

}

} else if (allToken.get(tokenPtr).getSt() == SymType.WHI) {

//<当循环语句>::=while<条件>do<语句>

int pos1 = allPcode.getPcodePtr();

tokenPtr++;

condition();

if (allToken.get(tokenPtr).getSt() == SymType.DO) {

int pos2 = allPcode.getPcodePtr();

allPcode.gen(Operator.JPC, 0, 0);

tokenPtr++;

statement();

allPcode.gen(Operator.JMP, 0, pos1);

allPcode.getAllPcode().get(pos2).setA(allPcode.getPcodePtr());

} else {

errorHandle(9, "");

return;

}

} else if (allToken.get(tokenPtr).getSt() == SymType.CAL) {

//<过程调用语句>::=call<标识符>

tokenPtr++;

int count = 0; //参数数目

PerSymbol tmp;

if (allToken.get(tokenPtr).getSt() == SymType.SYM) {

String name = allToken.get(tokenPtr).getValue();

if (allSymbol.isPreExists(name, level)) {

tmp = allSymbol.getSymbol(name);

if (tmp.getType() == allSymbol.getProc()) {

allPcode.gen(Operator.CAL, level - tmp.getLevel(), tmp.getValue());;

} else {

errorHandle(11, "");

return;

}

} else { //不存在该过程

errorHandle(10, "");

return;

}

tokenPtr++;

} else {

errorHandle(1, "");

return;

}

} else if (allToken.get(tokenPtr).getSt() == SymType.REA) {

//<读语句>::=read'('<标识符>{,<标识符>}')'

tokenPtr++;

if (allToken.get(tokenPtr).getSt() == SymType.LBR) {

tokenPtr++;

if (allToken.get(tokenPtr).getSt() == SymType.SYM) {

String name = allToken.get(tokenPtr).getValue();

if (!allSymbol.isPreExists(name, level)) {

errorHandle(10, "");

return;

} else {

PerSymbol tmp = allSymbol.getSymbol(name);

if (tmp.getType() == allSymbol.getVar()) {

allPcode.gen(Operator.OPR, 0, 16);

allPcode.gen(Operator.STO, level - tmp.getLevel(), tmp.getAddress());

} else {

errorHandle(12, "");

return;

}

}

}

tokenPtr++;

while (allToken.get(tokenPtr).getSt() == SymType.COMMA) {

tokenPtr++;

if (allToken.get(tokenPtr).getSt() == SymType.SYM) {

String name = allToken.get(tokenPtr).getValue();

if (!allSymbol.isPreExists(name, level)) {

errorHandle(10, "");

return;

} else {

PerSymbol tmp = allSymbol.getSymbol(name);

if (tmp.getType() == allSymbol.getVar()) {

allPcode.gen(Operator.OPR, 0, 16);

allPcode.gen(Operator.STO, level - tmp.getLevel(), tmp.getAddress());

} else {

errorHandle(12, "");

return;

}

}

tokenPtr++;

} else {

errorHandle(1, "");

return;

}

}

if (allToken.get(tokenPtr).getSt() == SymType.RBR) {

tokenPtr++;

} else {

errorHandle(5, "");

return;

}

} else {

errorHandle(4, "");

return;

}

} else if (allToken.get(tokenPtr).getSt() == SymType.WRI) {

//<写语句>::=write '('<表达式>{,<表达式>}')'

tokenPtr++;

if (allToken.get(tokenPtr).getSt() == SymType.LBR) {

tokenPtr++;

expression();

allPcode.gen(Operator.OPR, 0, 14);

while (allToken.get(tokenPtr).getSt() == SymType.COMMA) {

tokenPtr++;

expression();

allPcode.gen(Operator.OPR, 0, 14);

}

allPcode.gen(Operator.OPR, 0, 15);

if (allToken.get(tokenPtr).getSt() == SymType.RBR) {

tokenPtr++;

} else { //缺少)

errorHandle(5, "");

return;

}

} else { //缺少（

errorHandle(4, "");

}

} else if (allToken.get(tokenPtr).getSt() == SymType.BEG) {

//<复合语句>::=begin<语句>{;<语句>}end

body();

} else if (allToken.get(tokenPtr).getSt() == SymType.SYM) {

//<赋值语句>::=<标识符>:=<表达式>

String name = allToken.get(tokenPtr).getValue();

tokenPtr++;

if (allToken.get(tokenPtr).getSt() == SymType.CEQU || allToken.get(tokenPtr).getSt() == SymType.EQU || allToken.get(tokenPtr).getSt() == SymType.COL) {

if (allToken.get(tokenPtr).getSt() == SymType.EQU || allToken.get(tokenPtr).getSt() == SymType.COL) {

errorHandle(3, "");

}

tokenPtr++;

expression();

if (!allSymbol.isPreExists(name, level)) {

errorHandle(14, name);

return;

} else {

PerSymbol tmp = allSymbol.getSymbol(name);

if (tmp.getType() == allSymbol.getVar()) {

allPcode.gen(Operator.STO, level - tmp.getLevel(), tmp.getAddress());

} else {

errorHandle(13, name);

return;

}

}

} else {

errorHandle(3, "");

return;

}

} else if (allToken.get(tokenPtr).getSt() == SymType.REP) {

//<重复语句> ::= repeat<语句>{;<语句>}until<条件>

tokenPtr++;

int pos = allPcode.getPcodePtr();

statement();

while (allToken.get(tokenPtr).getSt() == SymType.SEMIC || isHeadOfStatement()) {

if (isHeadOfStatement()) {

errorHandle(1, "");

} else {

tokenPtr++;

}

if (allToken.get(tokenPtr).getSt() == SymType.UNT) {

errorHandle(22, "");

break;

}

tokenPtr++;

statement();

}

if (allToken.get(tokenPtr).getSt() == SymType.UNT) {

tokenPtr++;

condition();

allPcode.gen(Operator.JPC, 0, pos);

} else {

errorHandle(19, "");

return;

}

} else {

errorHandle(1, "");

return;

}

}

private void condition() {

//<条件>::=<表达式><关系运算符><表达式> | odd<表达式>

if (allToken.get(tokenPtr).getSt() == SymType.ODD) {

allPcode.gen(Operator.OPR, 0, 6);

tokenPtr++;

expression();

} else {

expression();

SymType tmp = allToken.get(tokenPtr).getSt();

tokenPtr++;

expression();

if (tmp == SymType.EQU) { //两个结果是否相等

allPcode.gen(Operator.OPR, 0, 8);

} else if (tmp == SymType.NEQE) { //两个结果是否不等

allPcode.gen(Operator.OPR, 0, 9);

} else if (tmp == SymType.LES) { //小于

allPcode.gen(Operator.OPR, 0, 10);

} else if (tmp == SymType.LARE) { //大于等于

allPcode.gen(Operator.OPR, 0, 11);

} else if (tmp == SymType.LAR) { //大于

allPcode.gen(Operator.OPR, 0, 12);

} else if (tmp == SymType.LESE) { //小于等于

allPcode.gen(Operator.OPR, 0, 13);

} else { //不合法的比较运算符

errorHandle(2, "");

}

}

}

private void expression() {

//<表达式>::=[+|-]<项>{<加法运算符><项>}

//<加法运算符>::=+|-

SymType tmp = allToken.get(tokenPtr).getSt();

if (tmp == SymType.ADD || tmp == SymType.SUB) {

tokenPtr++;

}

term();

if (tmp == SymType.SUB) {

allPcode.gen(Operator.OPR, 0, 1);

}

while (allToken.get(tokenPtr).getSt() == SymType.ADD || allToken.get(tokenPtr).getSt() == SymType.SUB) {

tmp = allToken.get(tokenPtr).getSt();

tokenPtr++;

term();

if (tmp == SymType.ADD) {

allPcode.gen(Operator.OPR, 0, 2);

} else if (tmp == SymType.SUB) {

allPcode.gen(Operator.OPR, 0, 3);

}

}

}

private void term() {

//<项>::=<因子>{<乘法运算符><因子>}

//<乘法运算符>::=\*|/

factor();

while (allToken.get(tokenPtr).getSt() == SymType.MUL || allToken.get(tokenPtr).getSt() == SymType.DIV) {

SymType tmp = allToken.get(tokenPtr).getSt();

tokenPtr++;

factor();

if (tmp == SymType.MUL) {

allPcode.gen(Operator.OPR, 0, 4);

} else if (tmp == SymType.DIV) {

allPcode.gen(Operator.OPR, 0, 5);

}

}

}

private void factor() {

//<因子>::=<标识符> | <无符号整数> | '('<表达式>')'

if (allToken.get(tokenPtr).getSt() == SymType.CONST) {

allPcode.gen(Operator.LIT, 0, Integer.parseInt(allToken.get(tokenPtr).getValue()));

tokenPtr++;

} else if (allToken.get(tokenPtr).getSt() == SymType.LBR) {

tokenPtr++;

expression();

if (allToken.get(tokenPtr).getSt() == SymType.RBR) {

tokenPtr++;

} else { //缺少右括号

errorHandle(5, "");

}

} else if (allToken.get(tokenPtr).getSt() == SymType.SYM) {

String name = allToken.get(tokenPtr).getValue();

if (! allSymbol.isPreExists(name, level)) {

errorHandle(10, "");

return;

} else {

PerSymbol tmp = allSymbol.getSymbol(name);

if (tmp.getType() == allSymbol.getVar()) {

allPcode.gen(Operator.LOD, level - tmp.getLevel(), tmp.getAddress());

} else if (tmp.getType() == allSymbol.getCon()) {

allPcode.gen(Operator.LIT, 0, tmp.getValue());

} else {

errorHandle(12, "");

return;

}

}

tokenPtr++;

} else {

errorHandle(1, "");

return;

}

}

private boolean isHeadOfStatement() {

return (allToken.get(tokenPtr).getSt() == SymType.IF ||

allToken.get(tokenPtr).getSt() == SymType.WHI ||

allToken.get(tokenPtr).getSt() == SymType.CAL ||

allToken.get(tokenPtr).getSt() == SymType.REA ||

allToken.get(tokenPtr).getSt() == SymType.WRI ||

allToken.get(tokenPtr).getSt() == SymType.BEG ||

allToken.get(tokenPtr).getSt() == SymType.SYM ||

allToken.get(tokenPtr).getSt() == SymType.REP);

}

private void errorHandle(int k, String name) {

errorHappen = true;

String error = "";

switch(k) {

case -1: //常量定义不是const开头，变量定义不是var开头

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + "wrong token";

break;

case 0: //缺少分号

if (allToken.get(tokenPtr).getSt() == SymType.SYM) {

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + "Missing ; before " + allToken.get(tokenPtr).getValue();

} else {

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + "Missing ; before " + allToken.get(tokenPtr).getSt();

}

break;

case 1: //标识符不合法

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + "Identifier illegal";

break;

case 2: //不合法的比较符

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + "illegal compare symbol";

break;

case 3: //常量赋值没用=

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + "Const assign must be =";

break;

case 4: //缺少（

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + "Missing (";

break;

case 5: //缺少）

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + "Missind )";

break;

case 6: //缺少begin

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + "Missing begin";

break;

case 7: //缺少end

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + "Missing end";

break;

case 8: //缺少then

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + "Missing then";

break;

case 9: //缺少do

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + "Missing do";

break;

case 10: //call, write, read语句中，不存在标识符

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + "Not exist" + allToken.get(tokenPtr).getValue();

break;

case 11: //该标识符不是proc类型

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + allToken.get(tokenPtr).getValue() + "is not a procedure";

break;

case 12: //read, write语句中，该标识符不是var类型

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + allToken.get(tokenPtr).getValue() + "is not a variable";

break;

case 13: //赋值语句中，该标识符不是var类型

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + name + "is not a varible";

break;

case 14: //赋值语句中，该标识符不存在

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + "not exist " + name;

break;

case 15: //该标识符已存在

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + "Already exist " + name;

break;

case 16: //调用函数参数错误

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + "Number of parameters of procedure " + name + "is incorrect";

break;

case 17: //缺少.

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + "Missing .";

break;

case 18: //多余代码

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + "too much code after .";

break;

case 19: //缺少until

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + "Missing until";

break;

case 20: //赋值符应为：=

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + "Assign must be :=";

break;

case 21: //end前多了；

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + "; is no need before end";

break;

case 22: //until前多了；

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + "; is no need before ubtil";

break;

case 23: //缺少,

error = "Error happened in line " + allToken.get(tokenPtr).getLine() + ":" + "Missing ,";

break;

}

errorMessage.add(error);

}

public List<String> getErrorMessage() {

return errorMessage;

}

public void showAllToken() {

for (int i = 0; i < allToken.size(); i++) {

System.out.println(allToken.get(i).getSt() + " " + allToken.get(i).getLine() + " " + allToken.get(i).getValue());

}

}

public List<Token> getAllToken() {

return allToken;

}

public void showAllSymbol() {

List<PerSymbol> display = allSymbol.getAllSymbol();

for (int i = 0; i < display.size(); i++) {

System.out.println(display.get(i).getType() + " " +

display.get(i).getName() + " " +

display.get(i).getValue() + " " +

display.get(i).getLevel() + " " +

display.get(i).getAddress());

}

}

public List<PerSymbol> getAllSymbol() {

return allSymbol.getAllSymbol();

}

public void showAllPcode() {

List<PerPcode> display = allPcode.getAllPcode();

for (int i = 0; i < display.size(); i++) {

System.out.print(i + " " + display.get(i).getF() + " ");

System.out.println(" " + display.get(i).getL() + " " +display.get(i).getA());

}

}

public List<PerPcode> getAllPcode() {

return allPcode.getAllPcode();

}

public List<String> interpreter(List<Integer> input) {

Interpreter one = new Interpreter(input);

one.setAllPcode(allPcode);

one.interpreter();

return one.getOutput();

}

public void interpreter() {

Interpreter one = new Interpreter();

one.setAllPcode(allPcode);

one.interpreter();

}

}

**Interpreter.java**

import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;

public class Interpreter {

private int STACK\_SIZE = 1000;

private int[] dataStack = new int[STACK\_SIZE];

private List<PerPcode> pcode;

private List<Integer> input;

private int inputPtr;

private List <String> output;

public Interpreter() {

}

public Interpreter(List<Integer> \_input) {

input = \_input;

output = new ArrayList<String>();

}

public void setAllPcode(AllPcode allPcode) {

pcode = allPcode.getAllPcode();

}

public List<String> getOutput() {

return output;

}

public void interpreter() {

int pc = 0; //程序计数器，指向下一条指令

int base = 0; //当前基地址

int top = 0; //程序运行栈栈顶

do {

PerPcode currentPcode = pcode.get(pc);

pc++;

if (currentPcode.getF() == Operator.LIT) {

//LIT：将常量送到运行栈S的栈顶，这时A段为常量值

dataStack[top] = currentPcode.getA();

top++;

} else if (currentPcode.getF() == Operator.OPR) {

//OPR：关系或算术运算，A段指明具体运算

switch(currentPcode.getA()) {

case 0:

//OPR 0 0 过程调用结束后,返回调用点并退栈

top = base;

pc = dataStack[base + 2];

base = dataStack[base];

break;

case 1:

//OPR 0 1 栈顶元素取反

dataStack[top - 1] = -dataStack[top - 1];

break;

case 2:

//OPR 0 2 次栈顶与栈顶相加，退两个栈元素，结果值进栈

dataStack[top - 2] = dataStack[top - 1] + dataStack[top - 2];

top--;

break;

case 3:

//OPR 0 3 次栈顶减去栈顶，退两个栈元素，结果值进栈

dataStack[top - 2] = dataStack[top - 2] - dataStack[top - 1];

top--;

break;

case 4:

//OPR 0 4 次栈顶乘以栈顶，退两个栈元素，结果值进栈

dataStack[top - 2] = dataStack[top - 1] \* dataStack[top - 2];

top--;

break;

case 5:

//OPR 0 5 次栈顶除以栈顶，退两个栈元素，结果值进栈

dataStack[top - 2] = dataStack[top - 2] / dataStack[top - 1];

top--;

break;

case 6:

//OPR 0 6 栈顶元素的奇偶判断，结果值在栈顶

dataStack[top - 1] = dataStack[top - 1] % 2;

break;

case 7:

break;

case 8:

//OPR 0 8 次栈顶与栈顶是否相等，退两个栈元素，结果值进栈

if (dataStack[top - 2] == dataStack[top - 1]) {

dataStack[top - 2] = 1;

} else {

dataStack[top - 2] = 0;

}

top--;

break;

case 9:

//OPR 0 9 次栈顶与栈顶是否不等，退两个栈元素，结果值进栈

if (dataStack[top - 2] != dataStack[top - 1]) {

dataStack[top - 2] = 1;

} else {

dataStack[top - 2] = 0;

}

top--;

break;

case 10:

//OPR 0 10 次栈顶是否小于栈顶，退两个栈元素，结果值进栈

if (dataStack[top - 2] < dataStack[top - 1]) {

dataStack[top - 2] = 1;

} else {

dataStack[top - 2] = 0;

}

top--;

break;

case 11:

//OPR 0 11 次栈顶是否大于等于栈顶，退两个栈元素，结果值进栈

if (dataStack[top - 2] >= dataStack[top - 1]) {

dataStack[top - 2] = 1;

} else {

dataStack[top - 2] = 0;

}

top--;

break;

case 12:

//OPR 0 12 次栈顶是否大于栈顶，退两个栈元素，结果值进栈

if (dataStack[top - 2] > dataStack[top - 1]) {

dataStack[top - 2] = 1;

} else {

dataStack[top - 2] = 0;

}

top--;

break;

case 13:

//OPR 0 13 次栈顶是否小于等于栈顶，退两个栈元素，结果值进栈

if (dataStack[top - 2] <= dataStack[top - 1]) {

dataStack[top - 2] = 1;

} else {

dataStack[top - 2] = 0;

}

top--;

break;

case 14:

//OPR 0 14 栈顶值输出至屏幕

System.out.print(dataStack[top - 1]);

System.out.print(" ");

//output.add(dataStack[top - 1] + " ");

break;

case 15:

//OPR 0 15 屏幕输出换行

System.out.println();

//output.add("\n");

break;

case 16:

//OPR 0 16 从命令行读入一个输入置于栈顶

System.out.println("please input a number");

Scanner s = new Scanner(System.in);

dataStack[top] = s.nextInt();

//dataStack[top] = input.get(inputPtr++);

top++;

break;

}

} else if (currentPcode.getF() == Operator.LOD) {

// LOD：将变量送到运行栈S的栈顶，这时A段为变量所在说明层中的相对位置。

dataStack[top] = dataStack[currentPcode.getA() + getBase(base, currentPcode.getL())];

top++;

} else if (currentPcode.getF() == Operator.STO) {

//STO：将运行栈S的栈顶内容送入某个变量单元中，A段为变量所在说明层中的相对位置。

dataStack[currentPcode.getA() + getBase(base, currentPcode.getL())] = dataStack[top - 1];

top--;

} else if (currentPcode.getF() == Operator.CAL) {

//CAL：调用过程，这时A段为被调用过程的过程体（过程体之前一条指令）在目标程序区的入口地址。

//跳转时，将该层基地址，跳转层基地址，pc指针保存在栈中

//基地址base变为此时栈顶top，pc指向要跳转的地方

//不修改top，因为前面代码已经将address+3，生成Pcode后会产生INT语句，修改top值

dataStack[top] = base;

dataStack[top + 1] = getBase(base, currentPcode.getL());

dataStack[top + 2] = pc;

base = top;

pc = currentPcode.getA();

} else if (currentPcode.getF() == Operator.INT) {

//INT：为被调用的过程（包括主过程）在运行栈S中开辟数据区，这时A段为所需数据单元个数（包括三个连接数据）；L段恒为0。

top = top + currentPcode.getA();

} else if (currentPcode.getF() == Operator.JMP) {

//JMP：无条件转移，这时A段为转向地址（目标程序）。

pc = currentPcode.getA();

} else if (currentPcode.getF() == Operator.JPC) {

//JPC：条件转移，当运行栈S的栈顶的布尔值为假（0）时，则转向A段所指目标程序地址；否则顺序执行。

if (dataStack[top - 1] == 0) {

pc = currentPcode.getA();

}

}

//System.out.println(pc + " " + base + " " + top);

} while (pc != 0);

}

//已知该层基地址为nowBp,得到层差为lev的层的基地址

private int getBase(int nowBp,int lev) {

int oldBp = nowBp;

while (lev > 0) {

oldBp = dataStack[oldBp + 1];

lev--;

}

return oldBp;

}

}

**LexAnalysis.java**

import java.io.BufferedReader;

import java.io.File;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.io.IOException;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* 词法分析程序

\*/

public class LexAnalysis {

private String[] keyWords = {

"begin", "end", "if", "then", "else", "const", "procedure", "var", "do", "while", "call", "read", "write", "odd", "repeat", "until"

};

private List<Token> allToken; //存放所有分析出来的token

private char ch = ' '; //当前字符

private int searchPtr = 0; //指向当前字符的指针

private char[] buffer; //存放所有源代码

private int line = 1; //当前行

private String strToken; //当前正在进行词法分析的字符串

public LexAnalysis(File file) {

init();

BufferedReader bf = null;

try {

bf = new BufferedReader(new FileReader(file));

String temp1 = "", temp2 = "";

while((temp1 = bf.readLine()) != null) {

temp2 = temp2 + temp1 + String.valueOf('\n');

}

buffer = temp2.toCharArray();

bf.close();

} catch (FileNotFoundException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

doAnalysis();

}

public List<Token> getAllToken() {

return allToken;

}

private void doAnalysis() {

while (searchPtr < buffer.length) {

allToken.add(analysis());

}

}

private Token analysis() {

strToken = "";

getChar();

while ((ch == ' ' || ch == '\n' || ch == '\t' || ch == '\0') && searchPtr < buffer.length) {

if (ch == '\n') {

line++;

}

getChar();

}

if (ch == '$' && searchPtr >= buffer.length) { //到达文件末尾

return new Token(SymType.EOF, line, "-1");

}

if (isLetter()) { //首位为字母，可能为保留字或者变量名

while (isLetter() || isDigit()) {

strToken += ch;

getChar();

}

retract();

for (int i = 0; i < keyWords.length; i++) {

if (strToken.equals(keyWords[i])) { //说明是保留字

return new Token(SymType.values()[i], line, "-");

}

}

//不是保留字，则为标识符，需要保存值

return new Token(SymType.SYM, line, strToken);

} else if (isDigit()) { //首位为数字，即为整数

while (isDigit()) {

strToken += ch;

getChar();

}

retract();

return new Token(SymType.CONST, line, strToken);

} else if (ch == '=') { //等号

return new Token(SymType.EQU, line, "-");

} else if (ch == '+') { //加号

return new Token(SymType.ADD, line, "-");

} else if (ch == '-') { //减号

return new Token(SymType.SUB, line, "-");

} else if (ch == '\*') { //乘号

return new Token(SymType.MUL, line, "-");

} else if (ch == '/') { //除号

return new Token(SymType.DIV, line, "-");

} else if (ch == '<') { //小于或不等于或小于等于

getChar();

if (ch == '=') {

return new Token(SymType.LESE, line, "-");

} else if (ch == '>') {

return new Token(SymType.NEQE, line, "-");

} else {

retract();

return new Token(SymType.LES, line, "-");

}

} else if (ch == '>') { //大于或大于等于

getChar();

if (ch == '=') {

return new Token(SymType.LARE, line, "-");

} else {

retract();

return new Token(SymType.LAR, line, "-");

}

} else if (ch == ',') { //逗号

return new Token(SymType.COMMA, line, "-");

} else if (ch == ';') { //分号

return new Token(SymType.SEMIC, line, "-");

} else if (ch == '.') { //点

return new Token(SymType.POI, line, "-");

} else if (ch == '(') { //左括号

return new Token(SymType.LBR, line, "-");

} else if (ch == ')') { //右括号

return new Token(SymType.RBR, line, "-");

} else if (ch == ':') { //赋值号

getChar();

if (ch == '=') {

return new Token(SymType.CEQU, line, "-");

} else {

retract();

return new Token(SymType.COL, line, "-");

}

}

return new Token(SymType.EOF, line, "-");

}

private void init() {

allToken = new ArrayList<Token>();

}

private char getChar() {

if (searchPtr < buffer.length) {

ch = buffer[searchPtr];

searchPtr++;

} else {

ch = '$';

}

return ch;

}

private void retract() {

searchPtr--;

ch = ' ';

}

private boolean isLetter() {

if(Character.isLetter(ch)) {

return true;

}

return false;

}

private boolean isDigit() {

if(Character.isDigit(ch)) {

return true;

}

return false;

}

}

**Operator.java**

/\*\*

\*这是Pcode的伪操作符集合

\*/

**public** **enum** Operator {

***INT***, ***CAL***, ***LIT***, ***LOD***, ***STO***, ***JMP***, ***JPC***, ***OPR***;

}

**PerPcode.java**

/\*\*

\* 单句Pcode语句类

\*/

**public** **class** PerPcode {

**private** Operator F;

**private** **int** L;

**private** **int** A;

PerPcode(Operator \_F, **int** \_L, **int** \_A) {

F = \_F;

L = \_L;

A = \_A;

}

**public** **void** setF(Operator \_F) {

F = \_F;

}

**public** **void** setL(**int** \_L) {

L = \_L;

}

**public** **void** setA(**int** \_A) {

A = \_A;

}

**public** Operator getF() {

**return** F;

}

**public** **int** getL() {

**return** L;

}

**public** **int** getA() {

**return** A;

}

}

**PerSymbol.java**

/\*\*

\* PerSymbol类

\*/

**public** **class** PerSymbol {

**private** **int** type; //表示常量、变量或过程

**private** **int** value; //表示常量或变量的值

**private** **int** level; //嵌套层次

**private** **int** address; //相对于所在嵌套过程基地址的地址

**private** **int** size; //表示常量，变量，过程所占的大小(这一项其实默认为0， 并没有用到)

**private** String name; //变量、常量或过程名

**public** PerSymbol(**int** \_type, **int** \_value, **int** \_level, **int** \_address, **int** \_size, String \_name) {

type = \_type;

value = \_value;

level = \_level;

address = \_address;

size = \_size;

name = \_name;

}

**public** PerSymbol(**int** \_type, **int** \_level, **int** \_address, **int** \_size, String \_name) {

//专为变量声明和过程声明写的构造函数

//变量和过程声明时没有初始值

type = \_type;

level = \_level;

address = \_address;

size = \_size;

name = \_name;

}

**public** **void** setType(**int** \_type) {

type = \_type;

}

**public** **void** setValue(**int** \_value) {

value = \_value;

}

**public** **void** setLevel(**int** \_level) {

level = \_level;

}

**public** **void** setAddress(**int** \_address) {

address = \_address;

}

**public** **void** setSize(**int** \_size) {

size = \_size;

}

**public** **void** setName(String \_name) {

name = \_name;

}

**public** **int** getType() {

**return** type;

}

**public** **int** getValue() {

**return** value;

}

**public** **int** getLevel() {

**return** level;

}

**public** **int** getAddress() {

**return** address;

}

**public** **int** getSize() {

**return** size;

}

**public** String getName() {

**return** name;

}

}

**SymType.java**

/\*\*

\*token类型

\*/

**public** **enum** SymType {

***BEG***, ***END***, ***IF***, ***THEN***, ***ELS***, ***CON***, ***PROC***, ***VAR***, ***DO***, ***WHI***, ***CAL***, ***REA***, ***WRI***, ***ODD***, ***REP***, ***UNT***,

//begin, end, if, then, else, const, procedure, var, do, while, call, read, write, odd, repeat, until

***EQU***, ***LES***, ***LESE***, ***LARE***, ***LAR***, ***NEQE***, ***ADD***, ***SUB***, ***MUL***, ***DIV***,

//=, <, <=, >=, >, <>, +, -, \*, /

***SYM***, ***CONST***,

//标识符， 常量

***CEQU***, ***COMMA***, ***SEMIC***, ***POI***, ***LBR***, ***RBR***,

//:=, ',' , ';', '.', '(', ')'

***COL***,

//:

***EOF***;

//end of file

}

**Token.java**

/\*\*

\*token类

\*/

**public** **class** Token {

**private** SymType st; //token的类别

**private** **int** line; //token所在行，错误处理使用

**private** String value; //token的值，只有标识符和常量有值

**public** Token(SymType \_st, **int** \_line, String \_value) {

st = \_st;

line = \_line;

value = \_value;

}

**public** **void** setSt(SymType \_st) {

st = \_st;

}

**public** **void** setLine(**int** \_line) {

line = \_line;

}

**public** **void** setValue(String \_value) {

value = \_value;

}

**public** SymType getSt() {

**return** st;

}

**public** **int** getLine() {

**return** line;

}

**public** String getValue() {

**return** value;

}

}